Algorithms for Synthesizing Priorities in Component-based Systems

Chih-Hong Cheng*, Saddek Bensalem†, Yu-Fang Chen‡, Rongjie Yan§,
Barbara Jobstmann†, Harald Ruess¶, Christian Buckl¶, Alois Knoll*

*Department of Informatics, Technische Universität München, Germany
†Verimag Laboratory, Grenoble, France
‡Institute of Information Science, Academia Sinica, Taipei, Taiwan
§State Key Laboratory of Computer Science, ISCAS, Beijing, China
¶fortiss GmbH, Germany

http://www6.in.tum.de/~chengch/vissbip

Abstract

We present algorithms to synthesize component-based systems that are safe and deadlock-free using priorities, which define stateless-precedence between enabled actions. Our core method combines the concept of fault-localization (using safety-game) and fault-repair (using SAT for conflict resolution). For complex systems, we propose three complementary methods as preprocessing steps for priority synthesis, namely (a) data abstraction to reduce component complexities, (b) alphabet abstraction and ♯-deadlock to ignore components, and (c) automated assumption learning for compositional priority synthesis.

I. Introduction

Priorities [15] define *stateless-precedence relations between actions* available in component-based systems. They can be used to restrict the behavior of a system in order to avoid undesired states. They are particularly useful to avoid deadlock states (i.e., states in which all actions are disabled), because they do not introduce new deadlock states and therefore avoid creating new undesired states. Furthermore, due to their stateless property and the fact that they operate on the interface of a component, they are relatively easy to implement in a distributed setting [17], [9]. In a tool paper [10], we presented the tool VissBIP[1] together with a concept called *priority synthesis*, which aims to automatically generate a set of priorities such that the system constrained by the synthesized priorities satisfies a given *safety property* or *deadlock freedom*. In this paper, we explain the underlying algorithm and propose extensions for more complex systems.

Priority synthesis is expensive; we showed in [11] that synthesizing priorities for safety properties (or deadlock-freedom) is NP-complete in the size of the state space of the product graph. Therefore, we present an incomplete search framework for priority synthesis, which mimics the process of *fault-localization* and *fault-repair* (Section III). Intuitively, a state is a fault location if it is the latest point from which there is a way to avoid a failure, i.e., there exists (i) an outgoing action that leads to an *attracted state*, a state from which all paths unavoidably reach a bad state, and (ii) there exists an alternative action that avoids entering any of the attracted states. We compute fault locations using the algorithm for *safety games*. Given a set of fault locations, priority synthesis is achieved via fault-repair: an algorithm resolves potential conflicts in priorities generated via fault-localization and finds a satisfying subset of priorities as a solution for synthesis. Our symbolic encodings on the system, together with the new variable ordering heuristic and other optimizations, helps to solve problems much more efficiently compared to our preliminary implementation in [10]. Furthermore, it allows us to integrate an adversary environment model similar to the setting in Ramadge and Wonham's controller synthesis framework [22].

Abstraction or compositional techniques are widely used in verification of infinite state or complex systems for safety properties but *not all* techniques ensure that synthesizing an abstract system for deadlock-freeness guarantees deadlock-freeness in the concrete system (Section IV). Therefore, it is important to find appropriate techniques to assist synthesis on complex problems. We first revisit *data abstraction* (Section IV-A) for data domain such that priority synthesis works on an abstract system composed by components abstracted component-wise [7]. Second, we present a technique called *alphabet-abstraction* (Section IV-B), handling complexities induced by the composition of components. Lastly, for behavioral-safety properties (not applicable for deadlock-avoidance), we utilize automata-learning [3] to achieve *compositional priority synthesis* (Section V).

We implemented the presented algorithms (except connection with the data abstraction module in D-Finder [8]) in the VissBIP tool and performed experiments to evaluate them (Section VI). Our examples show that the process using fault-localization and fault-repair generates priorities that are highly desirable. Alphabet abstraction enables us to scale to arbitrary large problems. We also present a model for distributed communication. In this example, the priorities synthesized by our engine are completely local (i.e., each priority involves two local actions within a component).

[1]Shortcut for **Vi**sualization and **s**ynthesis for **s**imple **BIP** systems.

Therefore, they can be translated directly to distributed control. We summarize related work and conclude with an algorithmic flow in Section VII and VIII.

II. Component-based Modeling and Priority Synthesis

A. Behavioral-Interaction-Priority Framework

The Behavior-Interaction-Priority (BIP) framework[2] provides a rigorous component-based design flow for heterogeneous systems. Rigorous design refers to the strict separation of three different layers (behaviors, interactions, and priorities) used to describe a system. A detailed description of the BIP language can be found in [6]. To simplify the explanations, we focus on *simple* systems, i.e., systems without hierarchies and finite data types. Intuitively, a simple BIP system consists of a set of automata (extended with data) that synchronize on joint labels.

Definition 1 (BIP System): We define a (simple BIP) system as a tuple $\mathcal{S} = (C, \Sigma, \mathcal{P})$, where

- Σ is a finite set of **events** or interaction labels, called **interaction alphabet**,
- $C = \bigcup_{i=1}^{m} C_i$ is a finite set of **components**. Each component C_i is a transition system extended with data. Formally, C_i is a tuple $(L_i, V_i, \Sigma_i, T_i, l_i^0, e_i^0)$:
  - $L_i = \{l_{i_1}, \ldots, l_{i_n}\}$ is a finite set of *control locations*.
  - $V_i = \{v_{i_1}, \ldots, v_{i_p}\}$ is a finite set of *(local) variables* with a finite domain. Wlog we assume that the domain is the Boolean domain $\mathbf{B} = \{\texttt{True}, \texttt{False}\}$. We use $|V_i|$ to denote the number of variables used in C_i. An *evaluation (or assignment)* of the variables in V_i is a functions $e : V_i \to \mathbf{B}$ mapping every variable to a value in the domain. We use $\mathcal{E}(V_i)$ to denote the set of all evaluations over the variables V_i. Given a Boolean formula $f \in \mathcal{B}(V_i)$ over the variables in V_i and an evaluation $e \in \mathcal{E}(V_i)$, we use $f(e)$ to refer to the truth value of f under the evaluation e.
  - $\Sigma_i \subseteq \Sigma$ is a subset of interaction labels used in C_i.
  - T_i is the set of *transitions*. A transition $t_i \in T_i$ is of the form (l, g, σ, f, l'), where $l, l' \in L_i$ are the *source and destination location*, $g \in \mathcal{B}(V_i)$ is called the *guard* and is a Boolean formula over the variables V_i. $\sigma \in \Sigma_i$ is an interaction label (specifying the event triggering the transition), and $f : V_i \to \mathcal{B}(V_i)$ is the *update function* mapping every variable to a Boolean formula encoding the change of its value.
  - $l_i^0 \in L_i$ is the *initial location* and $e_i^0 \in \mathcal{E}(V_i)$ is the initial evaluation of the variables.
- $\mathcal{P}$ is a finite set of interaction pairs (called **priorities**) defining a relation $\prec \subseteq \Sigma \times \Sigma$ between the interaction labels. We require that $\prec$ is (1) transitive and (2) non-reflexive (i.e., there are no circular dependencies) [15]. For $(\sigma_1, \sigma_2) \in \mathcal{P}$, we sometimes write $\sigma_1 \prec \sigma_2$ to highlight the property of priority.

Definition 2 (Configuration): Given a system $\mathcal{S}$, a *configuration (or state)* c is a tuple $(l_1, e_1, \ldots, l_m, e_m)$ with $l_i \in L_i$ and $e_i \in \mathcal{E}(V_i)$ for all $i \in \{1, \ldots, m\}$. We use $\mathcal{C}_\mathcal{S}$ to denote the set of all reachable configurations. The configuration $(l_1^0, e_1^0, \ldots, l_m^0, e_m^0)$ is called the *initial configuration* of $\mathcal{S}$ and is denoted by c^0.

Definition 3 (Enabled Interactions): Given a system $\mathcal{S}$ and a configuration $c = (l_1, e_1, \ldots, l_m, e_m)$, we say an interaction $\sigma \in \Sigma$ is **enabled (in** c**)**, if the following conditions hold:

1) (Joint participation) $\forall i \in \{1, \ldots, m\}$, if $\sigma \in \Sigma_i$, then $\exists g_i, f_i, l_i'$ such that $(l_i, g_i, \sigma, f_i, l_i') \in T_i$ and $g_i(e_i) = \texttt{True}$.
2) (No higher priorities enabled) For all other interaction $\bar{\sigma} \in \Sigma$ satisfying joint participation (i.e., $\forall i \in \{1, \ldots, m\}$, if $\bar{\sigma} \in \Sigma_i$, then $\exists (l_i, \bar{g}_i, \bar{\sigma}, \bar{f}_i, \bar{l}_i') \in T_i$ such that $\bar{g}_i(e_i) = \texttt{True}$), $(\sigma, \bar{\sigma}) \notin \mathcal{P}$ holds.

Definition 4 (Behavior): Given a system $\mathcal{S}$, two configurations $c = (l_1, e_1, \ldots, l_m, e_m)$, $c' = (l_1', e_1', \ldots, l_m', e_m')$, and an interaction $\sigma \in \Sigma$ enabled in c, we say c' *is a σ-successor (configuration) of* c, denoted $c \xrightarrow{\sigma} c'$, if the following two conditions hold for all components $C_i = (L_i, V_i, \Sigma_i, T_i, l_i^0, e_i^0)$:

- (Update for participated components) If $\sigma \in \Sigma_i$, then there exists a transition $(l_i, g_i, \sigma, f_i, l_i') \in T_i$ such that $g_i(e_i) = \texttt{True}$ and for all variables $v \in V_i$, $e_i' = f_i(v)(e_i)$.
- (Stutter for idle components) Otherwise, $l_i' = l_i$ and $e_i' = e_i$.

Given two configurations c and c', we say c' *is reachable from* c with the interaction sequence $w = \sigma_1 \ldots \sigma_k$, denoted $c \xrightarrow{w} c'$, if there exist configurations $c_0, \ldots, c_k$ such that (i) $c_0 = c$, (ii) $c_k = c'$, and (iii) for all $i : 0 \leq i < k$, $c_i \xrightarrow{\sigma_{i+1}} c_{i+1}$. We denote the set of all configuration of $\mathcal{S}$ reachable from the initial configuration c^0 by $\mathcal{R}_\mathcal{S}$. The *language* of a system $\mathcal{S}$, denoted $\mathcal{L}(\mathcal{S})$, is the set $\{w \in \Sigma^* \mid \exists c' \in \mathcal{R}_\mathcal{S}$ such that $c^0 \xrightarrow{w} c'\}$. Note that $\mathcal{L}(\mathcal{S})$ describes the behavior of $\mathcal{S}$, starting from the initial configuration c^0.

In this paper, we adapt the following simplifications:

- We do not consider uncontrollable events (of the environment), since the BIP language is currently not supporting them. However, our framework would allow us to do so. More precisely, we solve priority synthesis using a game-theoretic version of controller synthesis [22], in which uncontrollability can be modeled. Furthermore, since we consider only safety properties, our algorithms can be easily adapted to handle uncontrollable events.
- We do not consider data transfer during the interaction, as it is merely syntactic rewriting over variables between different components.

[2]http://www-verimag.imag.fr/Rigorous-Design-of-Component-Based.html?lang=en

B. Priority Synthesis for Safety and Deadlock Freedom

Definition 5 (Risk-Configuration/Deadlock Safety): Given a system $\mathcal{S} = (C, \Sigma, \mathcal{P})$ and the set of *risk configuration* $\mathcal{C}_{risk} \subseteq \mathcal{C}_{\mathcal{S}}$ (also called *bad states*), the system is **safe** if the following conditions hold. (A system that is not safe is called **unsafe**.)

- **(Deadlock-free)** $\forall c \in \mathcal{R}_{\mathcal{S}}, \exists \sigma \in \Sigma, \exists c' \in \mathcal{R}_{\mathcal{S}} : c \xrightarrow{\sigma} c'$
- **(Risk-state-free)** $\mathcal{C}_{risk} \cap \mathcal{R}_{\mathcal{S}} = \emptyset$.

Definition 6 (Priority Synthesis): Given a system $\mathcal{S} = (C, \Sigma, \mathcal{P})$, and the set of risk configuration $\mathcal{C}_{risk} \subseteq \mathcal{C}_{\mathcal{S}}$, priority synthesis searches for a set of priorities $\mathcal{P}_+$ such that

- For $\mathcal{P} \cup \mathcal{P}_+$, the defined relation $\prec_{\mathcal{P} \cup \mathcal{P}_+} \subseteq \Sigma \times \Sigma$ is also (1) transitive and (2) non-reflexive.
- $(C, \Sigma, \mathcal{P} \cup \mathcal{P}_+)$ is safe.

Given a system $\mathcal{S}$, we define the size of $\mathcal{S}$ as the size of the product graph induced by $\mathcal{S}$, i.e, $|\mathcal{R}_{\mathcal{S}}| + |\Sigma|$. Then, we have the following result.

Theorem 1 (Hardness of priority synthesis [11]): Given a system $\mathcal{S} = (C, \Sigma, \mathcal{P})$, finding a set $\mathcal{P}_+$ of priorities such that $(C, \Sigma, \mathcal{P} \cup \mathcal{P}_+)$ is safe is NP-complete in the size of $\mathcal{S}$.

We briefly mention the definition of **behavioral safety**, which is a powerful notion to capture erroneous behavioral-patterns for the system under design.

Definition 7 (Behavioral Safety): Given a system $\mathcal{S} = (C, \Sigma, \mathcal{P})$ and a regular language $\mathcal{L}_{\neg P} \subseteq \Sigma^*$ called the *risk specification*, the system is **B-safe** if $\mathcal{L}(\mathcal{S}) \cap \mathcal{L}_{\neg P} = \emptyset$. A system that is not B-safe is called **B-unsafe**.

It is well-known that the problem of asking for behavioral safety can be reduced to the problem of risk-state freeness. More precisely, since $\mathcal{L}_{\neg P}$ can be represented by a finite automaton $\mathcal{A}_{\neg P}$ (the monitor), priority synthesis for behavioral safety can be reduced to priority synthesis in the synchronous product of the system $\mathcal{S}$ and $\mathcal{A}_{\neg P}$ with the goal to avoid any product state that has a final state of $\mathcal{A}_{\neg P}$ in the second component.

III. A Framework of Priority Synthesis based on Fault-Localization and Fault-Repair

In this section, we describe our symbolic encoding scheme, followed by presenting our priority synthesis mechanism using a fault-localization and repair approach.

A. System Encoding

Our symbolic encoding is inspired by the execution semantics of the BIP engine, which during execution, selects one of the enabled interactions and executes the interaction. In our engine, we mimic the process and create a two-stage transition: For each iteration,

- (Stage 0) The *environment* raises all enabled interactions.
- (Stage 1) Based on the raised interactions, the *controller* selects one enabled interaction (if there exists one) while respecting the priority, and updates the state based on the enabled interaction.

Given a system $\mathcal{S} = (C, \Sigma, \mathcal{P})$, we use the following sets of Boolean variables to encode $\mathcal{S}$:

- $\{stg, stg'\}$ is the *stage indicator* and its primed version.
- $\bigcup_{\sigma \in \Sigma} \{\sigma, \sigma'\}$ are the variables representing interactions and their primed version. We use the same letter for an interaction and the corresponding variable, because there is a one-to-one correspondence between them.
- $\bigcup_{i=1...m} Y_i \cup Y_i'$, where $Y_i = \{y_{i1}, \ldots, y_{ik}\}$ and $Y_i' = \{y_{i1}', \ldots, y_{ik}'\}$ are the variables and their primed version, respectively, used to encode the locations L_i. (We use a binary encoding, i.e., $k = \lceil log|L_i| \rceil$). Given a location $l \in L_i$, we use $enc(l)$ and $enc'(l)$ to refer to the encoding of l using Y_i and Y_i', respectively.
- $\bigcup_{i=1...m} \bigcup_{v \in V_i} \{v, v'\}$ are the variables of the components and their primed version.

Algorithm 1: Generate Stage-0 transitions

input : System $\mathcal{S} = (C, \Sigma, \mathcal{P})$
output: Stage-0 transition predicate $\mathcal{T}_{stage_0}$
begin
  for $\sigma \in \Sigma$ **do**
1 `let` predicate $P_\sigma := \texttt{True}$
  for $\sigma \in \Sigma$ **do**
    for $i = \{1, \ldots, m\}$ **do**
2 **if** $\sigma \in \Sigma_i$ **then** $P_\sigma := P_\sigma \wedge \bigvee_{(l,g,\sigma,f,l') \in T_i} (enc(l) \wedge g)$
  `let` predicate $\mathcal{T}_{stage_0} := stg \wedge \neg stg'$
  for $\sigma \in \Sigma$ **do**
3 $\mathcal{T}_{stage_0} := T_{stage_0} \wedge (\sigma' \leftrightarrow P_\sigma)$
  for $i = \{1, \ldots, m\}$ **do**
4 $\mathcal{T}_{stage_0} := T_{stage_0} \wedge \bigwedge_{y \in Y_i} y \leftrightarrow y' \wedge \bigwedge_{v \in V_i} v \leftrightarrow v'$
  `return` $\mathcal{T}_{stage_0}$

Algorithm 2: Generate Stage-1 transitions

input : System $\mathcal{S} = (C, \Sigma, \mathcal{P})$
output: Stage-1 transition predicate $\mathcal{T}_{stage_1}$
begin
  `let` predicate $\mathcal{T}_{stage_1} := \texttt{False}$
  for $\sigma \in \Sigma$ **do**
    `let` predicate $T_\sigma := \neg stg \wedge stg'$
    for $i = \{1, \ldots, m\}$ **do**
      if $\sigma \in \Sigma_i$ **then**
1 $T_\sigma := T_\sigma \wedge \bigvee_{(l,g,\sigma,f,l') \in T_i} (enc(l) \wedge g \wedge \sigma \wedge \sigma' \wedge enc'(l') \wedge \bigwedge_{v \in V_i} v' \leftrightarrow f(v))$
    for $\sigma' \in \Sigma, \sigma' \neq \sigma$ **do**
2 $T_\sigma := T_\sigma \wedge \sigma' = \texttt{False}$
    for $i = \{1, \ldots, m\}$ **do**
3 **if** $\sigma \notin \Sigma_i$ **then** $T_\sigma := T_\sigma \wedge \bigwedge_{y \in Y_i} y \leftrightarrow y' \wedge \bigwedge_{v \in V_i} v \leftrightarrow v'$
    $\mathcal{T}_{stage_1} := \mathcal{T}_{stage_1} \vee T_\sigma$
  for $\sigma_1 \prec \sigma_2 \in \mathcal{P}$ **do**
4 $\mathcal{T}_{stage_1} := \mathcal{T}_{stage_1} \wedge ((\sigma_1 \wedge \sigma_2) \rightarrow \neg {\sigma_1}')$
  `return` $\mathcal{T}_{stage_1}$

We use Algorithm 1 and 2 to create transition predicates $\mathcal{T}_{stage_0}$ and $\mathcal{T}_{stage_1}$ for Stage 0 and 1, respectively. Note that $\mathcal{T}_{stage_0}$ and $\mathcal{T}_{stage_1}$ can be merged but we keep them separately, in order to (1) have an easy and direct way to synthesize priorities, (2) allow expressing the freedom of the environment, and (3) follow the semantics of the BIP engine.

- In Algorithm 1, Line 2 computes for each interaction σ the predicate P_σ representing all the configurations in which σ is enabled in the current configuration. In Line 3, starting from the first interaction, $\mathcal{T}_{stage_0}$ is continuously refined by conjoining $\sigma' \leftrightarrow P_\sigma$ for each interaction σ, i.e., the variables σ' is true if and only if the interaction σ is enabled. Finally, Line 4 ensures that the system configuration does not change in stage 0.
- In Algorithm 2, Line 1, 2, 3 are used to create the transition in which interaction σ is executed (Line 2 ensures that only σ is executed; Line 3 ensures the stuttering move of unparticipated components). Given a priority $\sigma_1 \prec \sigma_2$, in configurations in which σ_1 and σ_2 are both enabled (i.e., $\sigma_1 \wedge \sigma_2$ holds), the conjunction with Line 4 removes the possibility to execute σ_1 when σ_2 is also available.

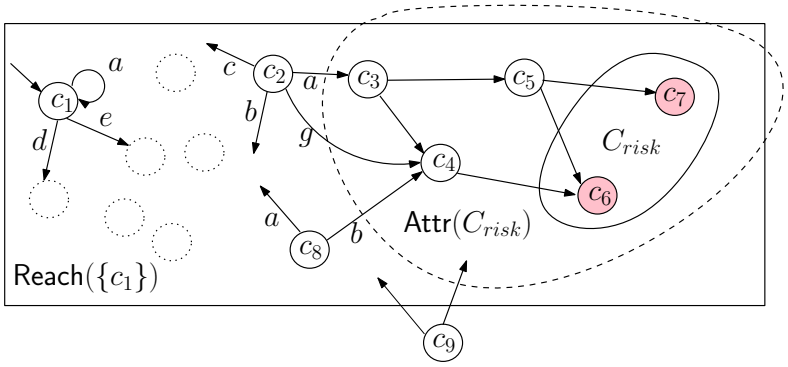

Figure 1. Locating fix candidates.

B. Step A. Finding Fix Candidates using Fault-localization

Synthesizing a set of priorities to make the system safe can be done in various ways, and we use Figure 1 to illustrate our underlying idea. Consider a system starting from state c_1. It has two risk configurations c_6 and c_7. In order to avoid risk using priorities, one method is to work on the initial configuration, i.e., to use the set of priorities $\{e \prec a, d \prec a\}$. Nevertheless, it can be observed that the synthesized result is not very desirable, as the behavior of the system has been greatly restricted.

Alternatively, our methodology works *backwards* from the set of risk states and finds states which is able to *escape from risk*. In Figure 1, as states c_3, c_4, c_5 unavoidably enter a risk state, they are within the *risk-attractor* ($\mathsf{Attr}(\mathcal{C}_{risk})$). For state c_2, c_8, and c_9, there exists an interaction which avoids risk. Thus, if a set of priorities $\mathcal{P}_+$ can ensure that from c_2, c_8, and c_9, the system can not enter the attractor, then $\mathcal{P}_+$ is the result of synthesis. Furthermore, as c_9 is not within the set of reachable states from the initial configuration ($\mathsf{Reach}(\{c_1\})$ in Figure 1), then it can be eliminated without consideration. We call $\{c_2, c_8\}$ a **fault-set**, meaning that an erroneous interaction can be taken to reach the risk-attractor.

Under our formulation, we can directly utilize the result of **algorithmic game solving** [16] to compute the fault-set. Algorithm 3 explains the underlying computation: For conciseness, we use $\exists\Xi$ ($\exists\Xi'$) to represent existential quantification over all unprimed (primed) variables used in the system encoding. Also, we use the operator $\texttt{SUBS}(X, \Xi, \Xi')$ for variable swap (substitution) from unprimed to primed variables in X: the `SUBS` operator is common in most BDD packages.

- In the beginning, we create P_{ini} for initial configuration, P_{dead} for deadlock (no interaction is enabled), and P_{risk} for risk configurations.
- In Part A, adding a stage-0 configuration can be computed similar to adding the environment state in a safety game. In a safety game, for an environment configuration to be added, there exists a transition which leads to the attractor.
- In Part A, adding a stage-1 configuration follows the intuition described earlier. In a safety game, for a control configuration c to be added, all outgoing transitions of c should lead to the attractor. This is captured by the set difference operation `PointTo` $\setminus$ `Escape` in Line 5.
- In Part B, Line 7 creates the transition predicate entering the attractor. Line 8 creates predicate `OutsideAttr` representing the set of stage-1 configuration outside the attractor. In Line 9, by conjuncting with `OutsideAttr` we ensure that the algorithm does not return a transition within the attractor.
- Part C removes transitions whose source is not within the set of reachable states.

Algorithm 3: Fault-localization

input : System $\mathcal{S} = (C, \Sigma, \mathcal{P})$, $\mathcal{T}_{stage_0}$, $\mathcal{T}_{stage_1}$
output: $\mathcal{T}_f \subseteq \mathcal{T}_{stage_1}$ as the set of stage-1 transitions starting from the fault-set but entering the risk attractor
begin

let $P_{ini} := stg \wedge \bigwedge_{i=1\ldots m}(enc(l_i^0) \wedge \bigwedge_{v \in V_i} v \leftrightarrow e_i^0(v))$
let $P_{dead} := \neg stg \wedge \bigwedge_{\sigma \in \Sigma} \neg\sigma$
let $P_{risk} := \neg stg \wedge \bigvee_{(l_1,e_1,\ldots,l_m,e_m) \in \mathcal{C}_{risk}} (enc(l_1) \wedge \bigwedge_{v \in V_1} v \leftrightarrow e_1(v) \wedge \ldots\ enc(l_m) \wedge \bigwedge_{v \in V_m} v \leftrightarrow e_m(v))$

`// Part A: solve safety game`
let $\text{Attr}_{pre} := P_{dead} \vee P_{risk}$, $\text{Attr}_{post} :=$ `False`
1 **while** `True` **do**
  `// add stage-0 (environment) configurations`
2 $\text{Attr}_{post,0} := \exists\Xi' : (\mathcal{T}_{stage_0} \wedge \texttt{SUBS}((\exists\Xi' : \text{Attr}_{pre}), \Xi, \Xi'))$
  `// add stage-1 (system) configurations`
3 **let** `PointTo` $:= \exists\Xi' : (\mathcal{T}_{stage_1} \wedge \texttt{SUBS}((\exists\Xi' : \text{Attr}_{pre}), \Xi, \Xi'))$
4 **let** `Escape` $:= \exists\Xi' : (\mathcal{T}_{stage_1} \wedge \texttt{SUBS}((\exists\Xi' : \neg\text{Attr}_{pre}), \Xi, \Xi'))$
5 $\text{Attr}_{post,1} :=$ `PointTo` $\setminus$ `Escape`
6 $\text{Attr}_{post} := \text{Attr}_{pre} \vee \text{Attr}_{post,0} \vee \text{Attr}_{post,1}$ `// Union the result`
  if $Attr_{pre} \leftrightarrow Attr_{post}$ **then** `break` `// Break when the image saturates`
  else $\text{Attr}_{pre} := \text{Attr}_{post}$

`// Part B: extract` $\mathcal{T}_f$
7 `PointTo` $:= \mathcal{T}_{stage_1} \wedge \texttt{SUBS}((\exists\Xi' : \text{Attr}_{pre}), \Xi, \Xi'))$
8 `OutsideAttr` $:= \neg\text{Attr}_{pre} \wedge (\exists\Xi' : \mathcal{T}_{stage_1})$
9 $\mathcal{T}_f :=$ `PointTo` $\wedge$ `OutsideAttr`

`// Part C: eliminate unused transition using reachable states`
let $\text{reach}_{pre} := P_{ini}$, $\text{reach}_{post} :=$ `False`
10 **while** `True` **do**
  $\text{reach}_{post} := \text{reach}_{pre} \vee \texttt{SUBS}(\exists\Xi : (\text{reach}_{pre} \wedge (\mathcal{T}_{stage_0} \vee \mathcal{T}_{stage_1})), \Xi', \Xi)$
  if $\text{reach}_{pre} \leftrightarrow \text{reach}_{post}$ **then** `break` `// Break when the image saturates`
  else $\text{reach}_{pre} := \text{reach}_{post}$
11 **return** $\mathcal{T}_f \wedge \text{reach}_{post}$

C. Step B. Priority Synthesis via Conflict Resolution - from Stateful to Stateless

Due to our system encoding, in Algorithm 3, the return value $\mathcal{T}_f$ contains not only the risk interaction but also all possible interactions simultaneously available. Recall Figure 1, $\mathcal{T}_f$ returns three transitions, and we can extract **priority candidates** from each transition.

- On c_2, a enters the risk-attractor, while b, g, c are also available. We have the following candidates $\{a \prec b, a \prec g, a \prec c\}$.
- On c_2, g enters the risk-attractor, while a, b, c are also available. We have the following candidates $\{g \prec b, g \prec c, g \prec a\}$[3].
- On c_8, b enters the risk-attractor, while a is also available. We have the following candidate $b \prec a$.

From these candidates, we can perform **conflict resolution** and generate a set of priorities that ensures avoiding the attractor. For example, $\{a \prec c, g \prec a, b \prec a\}$ is a set of satisfying priorities to ensure safety. Note that the set $\{a \prec b, g \prec b, b \prec a\}$ is not a legal priority set, because it creates circular dependencies. In our implementation, conflict resolution is performed using SAT solvers: In the SAT problem, any priority $\sigma_1 \prec \sigma_2$ is presented as a Boolean variable $\underline{\sigma_1 \prec \sigma_2}$, which can be set to `True` or `False`. If the generated SAT problem is satisfiable, for all variables $\underline{\sigma_1 \prec \sigma_2}$ which is evaluated to `True`, we add priority $\sigma_1 \prec \sigma_2$ to $\mathcal{P}_+$. The synthesis engine creates four types of clauses.

1) **[Priority candidates]** For each edge $t \in \mathcal{T}_f$ which enters the risk attractor using σ and having $\sigma_1, \ldots, \sigma_e$ available actions (excluding σ), create clause $(\bigvee_{i=1\ldots e} \underline{\sigma \prec \sigma_i})$[4].
2) **[Existing priorities]** For each priority $\sigma \prec \sigma' \in \mathcal{P}$, create clause $(\underline{\sigma \prec \sigma'})$.

[3]Notice that at least one candidate is a true candidate for risk-escape. Otherwise, during the attractor computation, c_2 will be included within the attractor.

[4]In implementation, Algorithm 3 works symbolically on BDDs and proceeds on **cubes** of the risk-edges (a cube contains a set of states having the same enabled interactions and the same risk interaction), hence it avoids enumerating edges state-by-state.

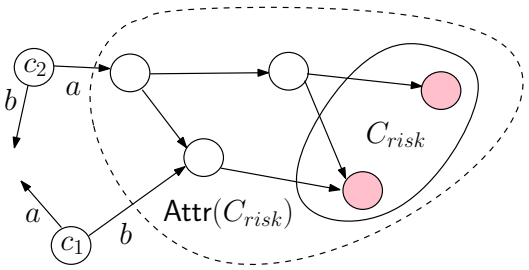

Figure 2. A simple scenario where conflicts are unavoidable on the fault-set.

3) **[Non-reflective]** For each interaction σ used in (1) and (2), create clause $(\neg\underline{\sigma \prec \sigma})$.
4) **[Transitive]** For any three interactions $\sigma_1, \sigma_2, \sigma_3$ used in (1) and (2), create clause $((\underline{\sigma_1 \prec \sigma_2} \wedge \underline{\sigma_2 \prec \sigma_3}) \Rightarrow \underline{\sigma_1 \prec \sigma_3})$.

When the problem is satisfiable, we only output the set of priorities within the priority candidates (as non-reflective and transitive clauses are inferred properties). Admittedly, here we still solve an NP-complete problem. Nevertheless,

- The number of interactions involved in the fault-set can be much smaller than Σ.
- As the translation does not involve complicated encoding, we observe from our experiment that solving the SAT problem does not occupy a large portion (less than 20% for all benchmarks) of the total execution time.

D. Optimization

Currently, we use the following optimization techniques compared to the preliminary implementation of [10].

1) Handling unsatisfiability: In the resolution scheme in Section III-C, when the generated SAT problem is unsatisfiable, we can redo the process by moving some states in the fault-set to the attractor. This procedure is implemented by selecting a subset of priority candidates and annotate to the original system. We call this process **priority-repushing**. E.g., consider the system $\mathcal{S} = (C, \Sigma, \mathcal{P})$ in Figure 2. The fault-set $\{c_1, c_2\}$ is unable to resolve the conflict: For c_1 the priority candidate is $a \prec b$, and for c_2 the priority candidate is $b \prec a$. When we redo the analysis with $\mathcal{S} = (C, \Sigma, \mathcal{P} \cup \{a \prec b\})$, this time c_2 will be in the attractor, as now c_2 must respect the priority and is unable to escape using a. Currently in our implementation, we supports the repushing under fixed depth to increase the possibility of finding a fix.

2) Initial Variable Ordering: Modified FORCE Heuristics: As we use BDDs to compute the risk-attractor, a good initial variable ordering can greatly influence the total required time solving the game. Although finding an optimal initial variable ordering is known to be NP-complete [23], many heuristics can be applied to find a good yet non-optimal ordering[5]. The basic idea of these heuristics is to group variables close if they participate in the same transition [13]; experiences have shown that this creates a BDD diagram of smaller size. Thus our goal is to find a heuristic algorithm which can be computed efficiently while creates a good ordering.

We adapt the concept in the FORCE heuristic [2]. Although the purpose of the FORCE heuristic is to work on SAT problems, we find the concept very beneficial in our problem setting. We explain the concept of FORCE based on the example in [2], and refer interested readers to the paper [2] for full details.

Given a CNF formula $C = c_1 \wedge c_2 \wedge c_3$, where $c_1 = (a \vee c), c_2 = (a \vee d), c_3 = (b \vee d)$.

- Consider a variable ordering $\langle a, b, c, d\rangle$. For this ordering, we try to evaluate it by considering the sum of the *span*. A span is the maximum distance between any two variables within the same clause. For c_1, under the ordering the span equals 2; for c_2 the span equals 3, and the sum of the span equals 7.
- Consider another variable ordering $\langle c, a, d, b\rangle$. Then the sum of span equals 3. Thus we consider that $\langle c, a, d, b\rangle$ is superior than $\langle a, b, c, d\rangle$.
- The purpose of the FORCE heuristic is to reduce the sum of such span. In the CNF example, the name of the heuristics suggests that a conceptual force representing each clause is grouping variables used within the clause.

Back to priority synthesis, consider the set of components $\bigcup_{i=1}^{n} C_i$ together with interaction labels Σ. We may similarly compute the sum of all spans, where now a span is *the maximum distance between any two components participating the same interaction* $\sigma \in \Sigma$. Precisely, we analogize clauses and variables in the original FORCE heuristic with interaction symbols and components. Therefore, we regard the FORCE heuristics equally applicable to create a better initial variable ordering for priority synthesis.

[Algorithm Sketch] Our modified FORCE heuristics is as follows.

1) Create an initial order of vertices composed from a set of components $\bigcup_{i=1}^{n} C_i$ and interactions $\sigma \in \Sigma$. Here we allow the user to provide an initial variable ordering, such that the FORCE heuristic can be applied more efficiently.

[5]Also, dynamic variable ordering, a technique which changes the variable ordering at run-time, can be beneficial when no good variable ordering is known [13]

2) Repeat for limited time or until the span stops decreasing:
   - Create an empty list.
   - For each interaction label $\sigma \in \Sigma$, derive its center of gravity $COG(\sigma)$ by computing the *average position* of all participated components. Use the average position as its value. Add the interaction with the value to the list.
   - For each component C_i, compute its value by $\frac{\sum_{\sigma \in Sigma_i} COG(\sigma)}{|\Sigma_i|}$. Add the component with the value to the list.
   - Sort the list based on the value. The resulting list is considered as a new variable ordering. Compute the new span and compare with the span from the previous ordering.

3) Dense variable encoding: The encoding in Section III-A is *dense* compared to the encoding in [10]. In [10], for each component C_i participating interaction σ, one separate variable σ_i is used. Then a joint action is done by an AND operation over all variables, i.e., $\bigwedge_i \sigma_i$. This eases the construction process but makes BDD-based game solving very inefficient: For a system $\mathcal{S}$, let $\Sigma_{use1} \subseteq \Sigma$ be the set of interactions where only one component participates within. Then the encoding in [10] uses at least $2|\Sigma \setminus \Sigma_{use1}|$ more BDD variables than the dense encoding.

4) Safety Engine Speedup: Lastly, as our created game graph is *bipartite*, Algorithm 3 can be refined to work on two separate images of stage-0 and stage-1, such that line 2 and line {3,4} are executed in alternation.

IV. Handling Complexities

In verification, it is standard to use *abstraction* and *modularity* to reduce the complexity of the analyzed systems. Abstraction is also useful in synthesis. However, note that if an abstract system is deadlock-free, it does not imply that the concrete system is as well. E.g., in Figure 3, the system composed by C_1 and C_2 contains deadlock (if both interactions a and b are required to be paired for execution). However, when we over-approximate C_1 to an abstract system C_1^α, a system composed by C_1^α and C_2 is deadlock free. On the other hand, deadlock-freeness of an under-approximation also does not imply deadlock-freeness of a concrete system. An obvious example can be obtained by under-approximating the system C_1 in Figure 3 to an abstract system C_1^β. Again, the composition of C_1^β and C_2 is deadlock-free, while the concrete system is not. Therefore, it is challenging to find a suitable abstract system such that the abstract system is deadlock-free implying that the concrete system is also deadlock-free.

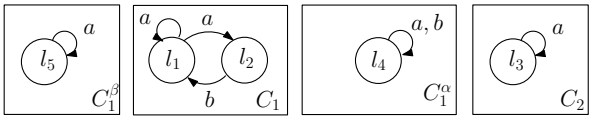

Figure 3. A scenario where the concrete system contains deadlock, but the abstract system is deadlock free.

In the following, we propose three techniques.

A. Data abstraction

Data abstraction techniques presented in the previous work [7] and implemented in the D-Finder tool kit [8] are *deadlock preserving*, i.e., synthesizing the abstract system to be deadlock free ensures that the concrete system is also deadlock free. Basically, the method works on an abstract system composed by components abstracted component-wise from concrete components. For example, if an abstraction preserves all control variables (i.e., all control variables are mapped by identity) and the mapping between the concrete and abstract system is precise with respect to all guards and updates (for control variables) on all transitions, then it is deadlock preserving. For further details, we refer interested readers to [7], [8].

B. Alphabet abstraction

Second, we present *alphabet abstraction*, targeting to synthesize priorities to avoid deadlock (but also applicable for risk-freeness with extensions). The underlying intuition is to abstract concrete behavior of components out of concern.

Definition 8 (Alphabet Transformer): Given a set Σ of interaction alphabet. Let $\Sigma_\Phi \subseteq \Sigma$ be **abstract alphabet**. Define $\alpha : \Sigma \to (\Sigma \setminus \Sigma_\Phi) \cup \{\sharp\}$ as the alphabet transformer, such that for $\sigma \in \Sigma$,

- If $\sigma \in \Sigma_\Phi$, then $\alpha(\sigma) := \sharp$.
- Otherwise, $\alpha(\sigma) := \sigma$.

Definition 9 (Alphabet Abstraction: Syntax): Given a system $\mathcal{S} = (C, \Sigma, \mathcal{P})$ and abstract alphabet $\Sigma_\Phi \subseteq \Sigma$, define the **$\sharp$-abstract system** $\mathcal{S}_\Phi$ to be $(C_\Phi, (\Sigma \setminus \Sigma_\Phi) \cup \{\sharp\}, \mathcal{P}_\Phi)$, where

- $C_\Phi = \bigcup_{i=1\ldots m} C_{i\Phi}$, where $C_{i\Phi} = (L_i, V_i, \Sigma_{i\Phi}, T_{i\Phi}, l_i^0, e_i^0)$ changes from C_i by **syntactically** replacing every occurrence of $\sigma \in \Sigma_i$ to $\alpha(\sigma)$.
- $\mathcal{P} = \bigcup_{i=1\ldots k} \sigma_i \prec \sigma_i'$ changes to $\mathcal{P}_\Phi = \bigcup_{i=1\ldots k} \alpha(\sigma_i) \prec \alpha(\sigma_i')$, and the relation defined by $\mathcal{P}_\Phi$ should be transitive and nonreflexive.

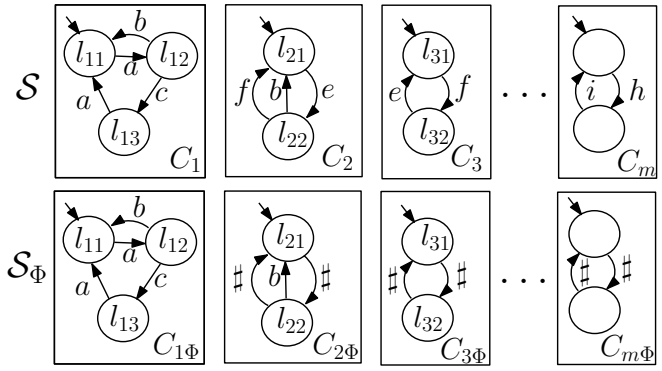

Figure 4. A system $\mathcal{S}$ and its $\sharp$-abstract system $\mathcal{S}_\Phi$, where $\Sigma_\Phi = \Sigma \setminus \{a, b, c\}$.

The definition for a configuration (state) of a $\sharp$-abstract system follows Definition 2. Denote the set of all configuration of $\mathcal{S}_\Phi$ reachable from c_0 as $\mathcal{C}_{\mathcal{S}_\Phi}$. The update of configuration for an interaction $\sigma \in \Sigma \setminus \Sigma_\Phi$ follows Definition 3. The only difference is within the semantics of the $\sharp$-interaction.

Definition 10 (Alphabet Abstraction: Semantics for $\sharp$-interaction): Given a configuration $c = (l_1, v_1, \ldots, l_m, v_m)$, the $\sharp$-interaction is **enabled** if the following conditions hold.

1) (≥ 1 participants) **Exists** $i \in \{1, \ldots, m\}$ where $\sharp \in \Sigma_{i\Phi}$, $\exists t_i = (l_i, g_i, \sharp, f_i, l_i') \in T_{i\Phi}$ such that $g(v_i) = \texttt{True}$.
2) (No higher priorities enabled) There exists no other interaction $\sigma_\flat \in \Sigma, (\sharp, \sigma_\flat) \in \mathcal{P}_\Phi$ such that $\forall i \in \{1, \ldots, m\}$ where $\sigma_\flat \in \Sigma_i$, $\exists t_{i\flat} = (l_i, g_{i\flat}, \sigma_{i\flat}, f_{i\flat}, l_i'') \in T_i$, $g_{i\flat}(v_i) = \texttt{True}$.

Then for a configuration $c = (l_1, v_1, \ldots, l_m, v_m)$, the configuration after taking an enabled $\sharp$-interaction changes to $c^\flat = (l_1^\flat, v_1^\flat, \ldots, l_m^\flat, v_m^\flat)$:

- (**May-update** for participated components) If $\sharp \in \Sigma_i$, then for transition $t_i = (l_i, g_i, \sharp, f_i, l_i') \in T_{i\Phi}$ such that $g_i(v_i) = \texttt{True}$, either
  1) $l_i^\flat = l_i'$, $v_i^\flat = f_i(v_i)$, or
  2) $l_i^\flat = l_i$, $v_i^\flat = v_i$.

  Furthermore, at least one component updates (i.e., select option 1).
- (Stutter for unparticipated components) If $\sharp \notin \Sigma_i$, $l_i^\flat = l_i$, $v_i^\flat = v_i$.

Lastly, the behavior of a $\sharp$-abstract system follows Definition 4. In summary, the above definitions indicate that in a $\sharp$-abstract system, any local transitions having alphabet symbols within Σ_Φ can be executed in isolation or jointly. Thus, we have the following result.

Lemma 1: Given a system $\mathcal{S}$ and its $\sharp$-abstract system $\mathcal{S}_\Phi$, define $\mathcal{R}_\mathcal{S}$ ($\mathcal{R}_{\mathcal{S}_\Phi}$) be the reachable states of system $\mathcal{S}$ (corresponding $\sharp$-abstract system) from from the initial configuration c^0. Then $\mathcal{R}_\mathcal{S} \subseteq \mathcal{R}_{\mathcal{S}_\Phi}$.

Proof: Result from the comparison between Definition 3 and 10. ■

As alphabet abstraction looses the execution condition by overlooking paired interactions, a $\sharp$-abstract system is deadlock-free does not imply that the concrete system is deadlock free. E.g., consider a system $\mathcal{S}'$ composed only by C_2 and C_3 in Figure 4. When $\Phi = \Sigma \setminus \{b\}$, its $\sharp$-abstract system $\mathcal{S}_\Phi'$ is shown below. In $\mathcal{S}'$, when C_2 is at location l_{21} and C_3 is at location l_{31}, interaction e and f are disabled, meaning that there exists a deadlock from the initial configuration. Nevertheless, in $\mathcal{S}_\Phi'$, as the $\sharp$-interaction is always enabled, it is deadlock free.

In the following, we strengthen the deadlock condition by the notion of **$\sharp$-deadlock**. Intuitively, a configuration is $\sharp$-deadlocked, if it is deadlocked, or the only interaction available is the $\sharp$-interaction.

Definition 11 ($\sharp$-deadlock): Given a $\sharp$-abstract system $\mathcal{S}_\Phi$, a configuration $c \in \mathcal{C}_{\mathcal{S}_\Phi}$ is $\sharp$-deadlocked, if $\nexists \sigma \in \Sigma \setminus \Sigma_\Phi, c' \in \mathcal{C}_{\mathcal{S}_\Phi}$ such that $c \xrightarrow{\sigma} c'$.

In other words, a configuration c of $\mathcal{S}_\Phi$ is $\sharp$-deadlocked implies that all interactions labeled with $\Sigma \setminus \Sigma_\Phi$ are disabled at c.

Lemma 2: Given a system $\mathcal{S}$ and its $\sharp$-abstract system $\mathcal{S}_\Phi$, define $\mathcal{D}$ as the set of deadlock states reachable from the initial state in $\mathcal{S}$, and $\mathcal{D}^\sharp$ as the set of $\sharp$-deadlock states reachable from the initial state in $\mathcal{S}_\Phi$. Then $\mathcal{D} \subseteq \mathcal{D}^\sharp$.

Proof: Consider a deadlock state $c \in \mathcal{D}$.

1) Based on Lemma 1, c is also in $\mathcal{R}_{\mathcal{S}_\Phi}$.
2) In $\mathcal{S}$, as $c \in \mathcal{D}$, all interactions are disabled in c. Then correspondingly in $\mathcal{S}_\Phi$, for state c, any interaction $\sigma \in \Sigma \setminus \Sigma_\Phi$ is also disabled. Therefore, c is $\sharp$-deadlocked.

Based on 1 and 2, $c \in \mathcal{D}^\sharp$. Thus $\mathcal{D} \subseteq \mathcal{D}^\sharp$. ■

Theorem 2: Given a system $\mathcal{S}$ and its $\sharp$-abstract system $\mathcal{S}_\Phi$, if $\mathcal{S}_\Phi$ is $\sharp$-deadlock-free, then $\mathcal{S}$ is deadlock-free.

Proof: As $\mathcal{S}_\Phi$ is $\sharp$-deadlock-free, we have $\mathcal{R}_{\mathcal{S}_\Phi} \cap \mathcal{D}^\sharp = \emptyset$. According to Lemma 1 and 2, we have $\mathcal{R}_\mathcal{S} \subseteq \mathcal{R}_{\mathcal{S}_\Phi}$ and $\mathcal{D} \subseteq \mathcal{D}^\sharp$. Hence $\mathcal{R}_\mathcal{S} \cap \mathcal{D} = \emptyset$, implying that $\mathcal{S}$ is deadlock-free. ■

(Algorithmic issues) Based on the above results, the use of alphabet abstraction and the notion of $\sharp$-deadlock offers a methodology for priority synthesis working on abstraction. Detailed steps are presented as follows.

1) Given a system $\mathcal{S}$, create its $\sharp$-abstract system $\mathcal{S}_\Phi$ by a user-defined $\Sigma_\Phi \subseteq \Sigma$. In our implementation, we let users select a subset of components $C_{s_1}, \ldots, C_{s_k} \in C$, and generate $\Sigma_\Phi = \Sigma \setminus (\Sigma_{s_1} \cup \ldots \cup \Sigma_{s_k})$.
   - E.g., consider system $\mathcal{S}$ in Figure 4 and its $\sharp$-abstract system $\mathcal{S}_\Phi$. The abstraction is done by looking at C_1 and maintaining $\Sigma_1 = \{a, b, c\}$.
   - When a system contains no variables, the algorithm proceeds by eliminateing components whose interaction are completely in the abstract alphabet. In Figure 4, as for $i = \{3 \ldots m\}$, $\Sigma_{i\Phi} = \{\sharp\}$, it is sufficient to eliminate all of them during the system encoding process.
2) If $\mathcal{S}_\Phi$ contains $\sharp$-deadlock states, we could obtain a $\sharp$-deadlock-free system by synthesizing a set of priorities $\mathcal{P}_+$, where the defined relation $\prec_+ \subseteq ((\Sigma \setminus \Sigma_\Phi) \cup \{\sharp\}) \times (\Sigma \setminus \Sigma_\Phi)$ using techniques presented in Section III.
   - In the system encoding, the predicate $P_{\sharp dead}$ for $\sharp$-deadlock is defined as $stg = \texttt{False} \wedge \bigwedge_{\sigma \in \Sigma \setminus \Sigma_\Phi} \sigma = \texttt{False}$.
   - If the synthesized priority is having the form $\sharp \prec \sigma$, then translate it into a set of priorities $\bigcup_{\sigma' \in \Sigma_\Phi} \sigma' \prec \sigma$.

V. Assume-guarantee Based Priority Synthesis

We use an assume-guarantee based compositional synthesis algorithm for behavior safety. Given a system $\mathcal{S} = (C_1 \cup C_2, \Sigma, \mathcal{P})$ and a risk specification described by a *deterministic finite state automaton* R, where $\mathcal{L}(R) \subseteq \Sigma^*$. We use $|\mathcal{S}|$ to denote the size of $\mathcal{S}$ and $|R|$ to denote the number of states of R. The synthesis task is to find a set of priority rules $\mathcal{P}_+$ such that adding $\mathcal{P}_+$ to the system $\mathcal{S}$ can make it B-Safe with respect to the risk specification $\mathcal{L}(R)$. This can be done using an *assume-guarantee* rule that we will describe in the next paragraph.

We first define some notations needed for the rule. The system $\mathcal{S}_+ = (C_1 \cup C_2, \Sigma, \mathcal{P} \cup \mathcal{P}_+)$ is obtained by adding priority rules $\mathcal{P}_+$ to the system $\mathcal{S}$. We use $\mathcal{S}_1 = (C_1, \Sigma, \mathcal{P} \cap \Sigma \times \Sigma_1)$ and $\mathcal{S}_2 = (C_2, \Sigma, \mathcal{P} \cap \Sigma \times \Sigma_2)$ to denote two sub-systems of $\mathcal{S}$. We further partition the alphabet Σ into three parts Σ_{12}, Σ_1, and Σ_2, where Σ_{12} is the set of interactions appear both in the sets of components C_1 and C_2 (in words, the shared alphabet of C_1 and C_2), Σ_i is the set of interactions appear only in the set of components C_i (in words, the local alphabet of C_i) for $i = 1, 2$. Also, we require that the decomposition of the system must satisfy that $\mathcal{P} \subseteq \Sigma \times (\Sigma_1 \cup \Sigma_2)$, which means that we do not allow a shared interaction to have a higher priority than any other interaction. This is **required** for the soundness proof of the assume-guarantee rule, as we also explained later that we will **immediately lose soundness by relaxing this restriction**. For $i = 1, 2$, the system $\mathcal{S}_{i+} = (C_i \cup \{d_i\}, \Sigma, (\mathcal{P} \cap \Sigma \times \Sigma_i) \cup \mathcal{P}_i)$ is obtained by (1) adding priority rules $\mathcal{P}_i \subseteq \Sigma \times \Sigma_i$ to $\mathcal{S}_i$ and, (2) in order to simulate stuttering transitions, adding a component d_i that contains only one location with self-loop transitions labeled with symbols in Σ_{3-i} (the local alphabet of the other set of components). Then the following assume-guarantee rule can be used to decompose the synthesis task into two smaller sub-tasks:

$$\begin{array}{rcll} \mathcal{L}(\mathcal{S}_{1+}) \cap \mathcal{L}(R) \cap \mathcal{L}(A) & = & \emptyset & (a) \\ \mathcal{L}(\mathcal{S}_{2+}) \cap \mathcal{L}(\overline{A}) & = & \emptyset & (b) \\ \hline \mathcal{L}(\mathcal{S}_{+}) \cap \mathcal{L}(R) & = & \emptyset & (c) \end{array}$$

The above assume-guarantee rule says that $\mathcal{S}_+$ is B-Safe with respect to $\mathcal{L}(R)$ iff there exists an assumption automaton A such that (1) $\mathcal{S}_{1+}$ is B-Safe with respect to $\mathcal{L}(R) \cap \mathcal{L}(A)$ and (2) $\mathcal{S}_{2+}$ is B-Safe with respect to $\mathcal{L}(\overline{A})$, where $\overline{A}$ is the complement of A, $\mathcal{P}_+ = \mathcal{P}_1 \cup \mathcal{P}_2$ and no conflict in $\mathcal{P}_1$ and $\mathcal{P}_2$. In the following, we prove the above assume-guarantee rule is both sound and complete. Nevertheless, it is unsound for deadlock freeness. An example can be found at the beginning of Section IV.

Theorem 3 (Soundness): Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be two non-conflicting priority rules, A be the assumption automaton, R be the risk specification automaton, $\mathcal{S}_{1+} = (C_1 \cup \{d_1\}, \Sigma, (\mathcal{P} \cap \Sigma \times \Sigma_1) \cup \mathcal{P}_1)$, and $\mathcal{S}_{2+} = (C_2 \cup \{d_2\}, \Sigma, (\mathcal{P} \cap \Sigma \times \Sigma_2) \cup \mathcal{P}_2)$, where $\mathcal{P}_i \subseteq \Sigma \times \Sigma_i$ for $i = 1, 2$ and $\mathcal{P} \subseteq \Sigma \times (\Sigma_1 \cup \Sigma_2)$. If $\mathcal{L}(\mathcal{S}_{1+}) \cap \mathcal{L}(R) \cap \mathcal{L}(A) = \emptyset$ and $\mathcal{L}(\mathcal{S}_{2+}) \cap \mathcal{L}(\overline{A}) = \emptyset$. The priority rule $\mathcal{P}_1 \cup \mathcal{P}_2$ ensures that the system $\mathcal{S} = (C_1 \cup C_2, \Sigma, \mathcal{P})$ is B-Safe with respect to R.

Proof: First, from $\mathcal{L}(\mathcal{S}_{1+}) \cap \mathcal{L}(R) \cap \mathcal{L}(A) = \emptyset$ and $\mathcal{L}(\mathcal{S}_{2+}) \cap \mathcal{L}(\overline{A}) = \emptyset$, we can obtain the relation between those languages described in Figure 5. From the figure, one can see that the two languages $\mathcal{L}(\mathcal{S}_{1+}) \cap \mathcal{L}(R)$ and $\mathcal{L}(\mathcal{S}_{2+})$ are disjoint. This follows that $\mathcal{L}(\mathcal{S}_{1+}) \cap \mathcal{L}(R) \cap \mathcal{L}(\mathcal{S}_{2+}) = \emptyset$. By Lemma 3, we have $\mathcal{L}(\mathcal{S}_+) \cap \mathcal{L}(R) \subseteq \mathcal{L}(\mathcal{S}_{1+}) \cap \mathcal{L}(\mathcal{S}_{2+}) \cap \mathcal{L}(R) = \emptyset$. Hence the set of priorities $\mathcal{P}_1 \cup \mathcal{P}_2$ ensures that $\mathcal{S}$ is B-Safe with respect to R. ■

Lemma 3 (Composition): Let $\mathcal{S}_1 = (C_1 \cup \{d_1\}, \Sigma, \mathcal{P}_1)$, and $\mathcal{S}_2 = (C_2 \cup \{d_2\}, \Sigma, \mathcal{P}_2)$, and $\mathcal{S}_{1+2} = (C_1 \cup C_2, \Sigma, \mathcal{P}_1 \cup \mathcal{P}_2)$ be three systems, where $\mathcal{P}_i \subseteq \Sigma \times \Sigma_i$ for $i = 1, 2$. We have $\mathcal{L}(\mathcal{S}_{1+2}) \subseteq \mathcal{L}(\mathcal{S}_1) \cap \mathcal{L}(\mathcal{S}_2)$.

Proof: For a word $w = \sigma_1, \ldots, \sigma_n \in \mathcal{L}(\mathcal{S}_{1+2})$, we consider inductively from the first interaction. If σ_1 is enabled in the initial configuration $(l_1, v_1, \ldots, l_n, v_n, \ldots l_m, v_m)$ of $\mathcal{S}_{1+2}$, then according to Definition 3, we have (1) if σ_1 is in the interaction alphabet of component $c_i \in C_1 \cup C_2$, then there exist a transition $(l_i, g_i, \sigma_1, f_i, l_i')$ in c_i such that $g_i(v_i) = \texttt{True}$ and (2) there exists no transition $(l_i, g_i, \sigma', f_i, l_i')$ in components of C_1 and C_2 such that $g_i(v_i) = \texttt{True}$ and $(\sigma_1, \sigma') \in \mathcal{P}_1 \cup \mathcal{P}_2$.

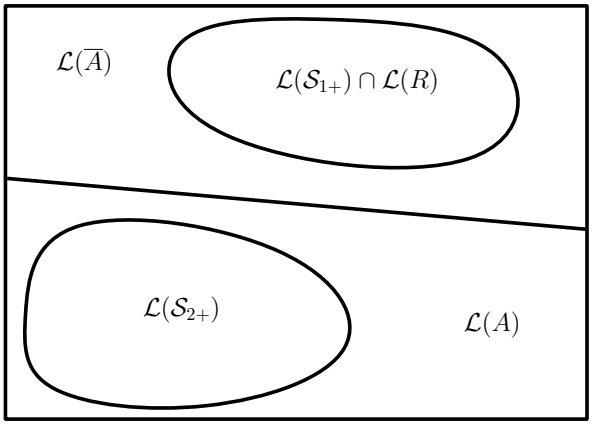

Figure 5. The relation between the languages.

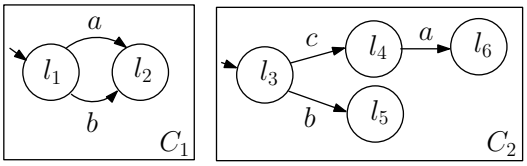

Figure 6. A counterexample when we allow a shared interaction to have higher priority than others.

We want to show that σ_1 is also enabled in the initial configuration of $\mathcal{S}_1$. In order to do this, we have to prove (1) components in $C_1 \cup \{d_1\}$ can move with σ_1 and (2) there exists no transition $(l_i, g_i, \sigma', f_i, l_i')$ in $C_1 \cup \{d_i\}$ such that $g_i(v_i) = \texttt{True}$, l_i is an initial location, and $(\sigma_1, \sigma') \in \mathcal{P}_1$.

- For (1), we consider the following cases: (a) If $\sigma_1 \in \Sigma_{12}$, components of C_1 can move with σ_1 and d_1 can move with σ_1 via a self-loop transition. (b) If $\sigma_1 \in \Sigma_1$, components of C_1 can move with σ_1 and it is not an interaction of d_1. (c) If $\sigma_1 \in \Sigma_2$, it is not an interaction of C_1 and d_1 can move with σ_1 via a self-loop transition. Therefore, components in $C_1 \cup \{d_1\}$ can move with σ_1.
- For (2), first, it is not possible to have such a transition in any component of C_1 by the definition of $\mathcal{S}_{1+2}$ and Definition 3. Then, if the transition is in d_i, we have $\sigma' \in \Sigma_2$ and it follows that $(\sigma, \sigma') \notin \mathcal{P}_1 \subseteq \Sigma \times \Sigma_1$.

By the above arguments for (1) and (2), σ_1 is enabled in the initial configuration of $\mathcal{S}_1$. By a similar argument, σ_1 is also enabled in the initial configuration of $\mathcal{S}_2$.

The inductive step can be proved using the same argument. Thus $w \in \mathcal{L}(\mathcal{S}_1)$ and $w \in \mathcal{L}(\mathcal{S}_2)$. It follows that $\mathcal{L}(\mathcal{S}_{1+2}) \subseteq \mathcal{L}(\mathcal{S}_1) \cap \mathcal{L}(\mathcal{S}_2)$. ■

Theorem 4 (Completeness): Let $\mathcal{S}_+ = (C, \Sigma, \mathcal{P} \cup \mathcal{P}_+)$ be a system and R be the risk specification automaton. If $\mathcal{L}(\mathcal{S}_+) \cap \mathcal{L}(R) = \emptyset$, then there exists an assumption automaton A, system components C_1 and C_2 such that $C = C_1 \cup C_2$, $C_1 \cap C_2 = \emptyset$, and two non-conflicting priority rules $\mathcal{P}_1 \subseteq \Sigma \times \Sigma_1$ and $\mathcal{P}_2 \subseteq \Sigma \times \Sigma_2$ such that $\mathcal{L}(C_1 \cup \{d_1\}, \Sigma, \mathcal{P} \cup \mathcal{P}_1) \cap \mathcal{L}(R) \cap \mathcal{L}(A) = \emptyset$, $\mathcal{L}(C_2 \cup \{d_2\}, \Sigma, \mathcal{P} \cup \mathcal{P}_2) \cap \mathcal{L}(\overline{A}) = \emptyset$, and $\mathcal{P}_+ = \mathcal{P}_1 \cup \mathcal{P}_2$.

Proof: Can be proved by taking $C_1 = C$, $C_2 = \emptyset$, A as an automaton that recognizes Σ^*, $\mathcal{P}_1 = \mathcal{P}_+$, and $\mathcal{P}_2 = \emptyset$. ■

Below we give an example that if we allow the priority $\mathcal{P}$ to be any relation between the interactions, then the assume-guarantee rule we used is unsound. The key is that Lemma 3 will no longer be valid with the relaxed constraints to the priority. In Figure 6, both C_1 and C_2 has only one components, $\Sigma_1 = \emptyset$, $\Sigma_2 = \{c\}$, and $\Sigma_{12} = \{a, b\}$. Assume that we have the priority rule $\mathcal{P} = \{b \prec a\}$ in $\mathcal{S}_1$, $\mathcal{S}_2$, and $\mathcal{S}$. Then we get $\mathcal{L}(\mathcal{S}_1) = \{a\}$, $\mathcal{L}(\mathcal{S}_2) = \{b + ca\}$, which implies $\mathcal{L}(\mathcal{S}_1) \cap \mathcal{L}(\mathcal{S}_2) = \emptyset$. However, $\mathcal{L}(\mathcal{S}) = \{b\}$. Then we found a counterexample for Lemma 3. This produces a counterexample of the soundness of the assume-guarantee rule. With a risk specification $\mathcal{L}(R) = \{b\}$, an assumption automaton $\mathcal{L}(A) = \Sigma^*$, and priorities $\mathcal{P} = \mathcal{P}_1 = \mathcal{P}_2 = \{b \prec a\}$, the subtasks of the assume-guarantee rule can be proved to be B-Safe. However, the system $\mathcal{S}$ is not B-Safe with respect to $\mathcal{L}(R)$. The reason why Σ_{12} can not be placed on the right-hand side of $\mathcal{P}$, $\mathcal{P}_1$, and $\mathcal{P}_2$ is because even in the subsystem a shared interaction can block other interactions successfully, when composing two systems together, it may no longer block other interactions (as now they need to be paired).

Notice that (1) the complexity of a synthesis task is NP-complete in the number of states in the risk specification automaton product with the size of the system and (2) $|\mathcal{S}|$ is approximately equals to $|\mathcal{S}_1| \times |\mathcal{S}_2|$[6]. Consider the case

[6]This is true only if the size of the alphabet is much smaller than the number of reachable configurations.

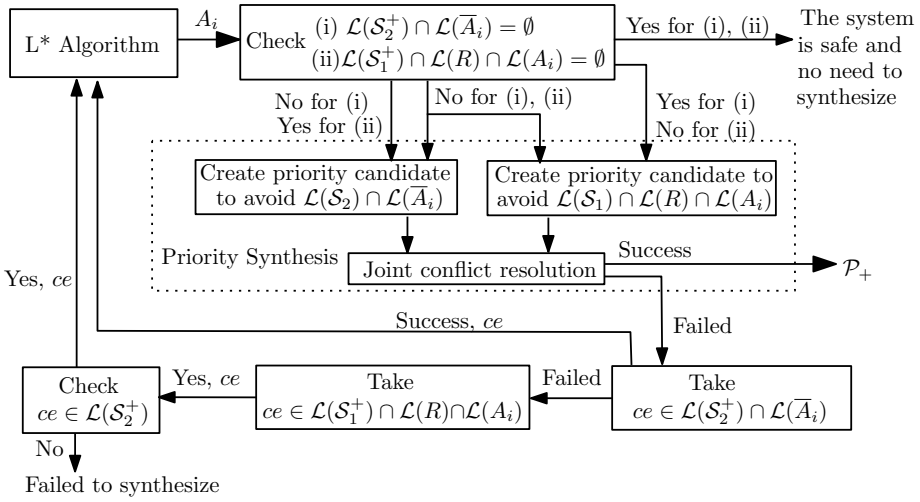

Figure 7. The flow of the assume-guarantee priority synthesis.

that one decomposes the synthesis task of $\mathcal{S}$ with respect to $\mathcal{L}(R)$ into two subtasks using the above assume-guarantee rule. The complexity original synthesis task is NP-complete in $|\mathcal{S}| \times |R|$ and the complexity of the two sub-tasks are $|\mathcal{S}_1| \times |R| \times |A|$ and $|\mathcal{S}_2| \times |A|$[7], respectively. Therefore, if one managed to find a small assumption automaton A for the assume-guarantee rule, the complexity of synthesis can be greatly reduced. We propose to use the machine learning algorithm L* [3] to automatically find a small automaton that is suitable for compositional synthesis. Next, we will first briefly describe the L* algorithm and then explain how to use it for compositional synthesis.

The L* algorithm works iteratively to find a minimal deterministic automaton recognizing a target regular language U. It assumes a *teacher* that answers two types of queries: (a) *membership queries* on a string w, where the teacher returns *true* if w is in U and *false* otherwise, (b) *equivalence queries* on an automaton A, where the teacher returns *true* if $\mathcal{L}(A) = U$, otherwise it returns *false* together with a counterexample string in the difference of $\mathcal{L}(A)$ and U. In the i-th iteration of the algorithm, the L* algorithm acquires information of U by posing membership queries and guess a candidate automaton A_i. The correctness of the A_i is then verified using an equivalence query. If A_i is not a correct automaton (i.e., $\mathcal{L}(A) \neq U$), the counterexample returned from the teacher will be used to refine the conjecture automaton of the $(i+1)$-th iteration. The learning algorithm is guaranteed to converge to the minimal deterministic finite state automaton of U in a polynomial number of iterations[8]. Also the sizes of conjecture automata increase strictly monotonically with respect to the number of iterations (i.e., $|A_{i+1}| > |A_i|$ for all $i > 0$).

The flow of our compositional synthesis is in Figure 7. Our idea of compositional synthesis via learning is the following. We use the notations $\mathcal{S}_i^+$ to denote the system $\mathcal{S}_i$ equipped with a stuttering component. First we use L* to learn the language $\mathcal{L}(\mathcal{S}_2^+)$. Since the transition system induced from the system $\mathcal{S}_2^+$ has finitely many states, one can see that $\mathcal{L}(\mathcal{S}_2^+)$ is regular. For a membership query on a word w, our algorithm simulates it symbolically on $\mathcal{S}_2^+$ to see if it is in $\mathcal{L}(\mathcal{S}_2^+)$. Once the L* algorithm poses an equivalence query on a deterministic finite automaton A_i, our algorithm tests conditions $\mathcal{L}(\mathcal{S}_1^+) \cap \mathcal{L}(R) \cap \mathcal{L}(A_i) = \emptyset$ and $\mathcal{L}(\mathcal{S}_2^+) \cap \mathcal{L}(\overline{A_i}) = \emptyset$ one after another. So far, our algorithm looks very similar to the compositional verification algorithm proposed in [14]. There are a few possible outcomes of the above test

1) Both condition holds and we proved the system is B-Safe with respect to $\mathcal{L}(R)$ and no synthesis is needed.
2) At least one of the two conditions does not hold. In such case, we try to synthesize priority rules to make the system B-Safe (see the details below).
3) If the algorithm fails to find usable priority rules, we have two cases:
   a) The algorithm obtains a counterexample string ce in $\mathcal{L}(\mathcal{S}_1^+) \cap \mathcal{L}(R) \setminus \mathcal{L}(\overline{A_i})$ from the first condition. This case is more complicated. We have to further test if $ce \in \mathcal{L}(\mathcal{S}_2^+)$. A negative answer implies that ce is in $\mathcal{L}(A_i) \setminus \mathcal{L}(\mathcal{S}_2^+)$. This follows that ce can be used by L* to refine the next conjecture. Otherwise, our algorithm terminates and reports not able to synthesize priority rules.
   b) The algorithm obtains a counterexample string ce in $\mathcal{L}(\mathcal{S}_2^+) \setminus \mathcal{L}(A_i)$ from the second condition, in such case, ce can be used by L* to refine the next conjecture.

The deterministic finite state automata R, A_i, and also its complement $\overline{A_i}$ can be treated as components without data and can be easily encoded symbolically using the approach in Section III-A. Also the two conditions can be tested using standard symbolic reachability algorithms.

[7]Since A is deterministic, the sizes of A and its complement $\overline{A}$ are identical.

[8]In the size of the minimal deterministic finite state automaton of U and the longest counterexample returned from the teacher.

Compositional Synthesis: Recall that our goal is to find a set of suitable priority rules via a small automaton A_i. Therefore, before using the ce to refine and obtain the next conjecture A_{i+1}, we first attempt to synthesis priority rules using A_i as the assumption automaton. Synthesis algorithms in previous sections can then be applied separately to the system composed of $\{\mathcal{S}_1^+, R, A_i\}$ and the system composed of $\{\mathcal{S}_2^+, \overline{A_i}\}$ to obtain two non-conflicting priority rules $\mathcal{P}_{1i} \subseteq (\Sigma_1 \cup \Sigma_{12}) \times \Sigma_1$ and $\mathcal{P}_{2i} \subseteq (\Sigma_2 \cup \Sigma_{12}) \times \Sigma_2$. Then $\mathcal{P}_{1i} \cup \mathcal{P}_{2i}$ is the desired priority for $\mathcal{S}$ to be B-Safe with respect to R. To be more specific, we first compute the CNF formulae f_1 and f_2 (that encode all possible priority rules that are *local*, i.e., we remove all non-local priority candidates) of the two systems separately using the algorithms in Section III, and then check satisfiability of $f_1 \wedge f_2$. The priority rules $\mathcal{P}_{1i}$ and $\mathcal{P}_{2i}$ can be derived from the satisfying assignment of $f_1 \wedge f_2$.

VI. Evaluation

Table I
Experimental results

	Time (seconds)				# of BDD variables				
Problem	NFM[1]	Opt.[2]	Ord.[3]	Abs.[4]	NFM	Opt.	Ord.	Abs.	Remark
Phil. 10	0.813	0.303	0.291	0.169	202	122	122	38	[1] Engine based on [10]
Phil. 20	-	86.646	0.755	0.166	-	242	242	38	[2] Dense var. encoding
Phil. 25	-	-	1.407	0.183	-	-	302	38	[3] Initial var. ordering
Phil. 30	-	-	3.740	0.206	-	-	362	38	[4] Alphabet abstraction
Phil. 35	-	-	5.913	0.212	-	-	422	38	- Timeout/Not evaluated
Phil. 40	-	-	10.210	0.228	-	-	482	38	
Phil. 45	-	-	18.344	0.213	-	-	542	38	
Phil. 50	-	-	30.384	0.234	-	-	602	38	
DPU v1	5.335	0.299	x	x	168	116	x	x	[R] Priority repushing
DPU v2	4.174	0.537	1.134[R]	x	168	116	116[R]	x	x Not evaluated
Traffic	x	x	0.651	x	x	x	272	x	

We implemented the presented algorithms (except connection the data abstraction module in D-Finder [8]) in the VissBIP[9] tool and performed experiments to evaluate them. To observe how our algorithm scales, in Table I we summarize results of synthesizing priorities for the dining philosophers problem[10]. Our preliminary result in [10] fails to synthesize priorities when the number of philosophers is greater than 15 (i.e., a total of 30 components), while currently we are able to solve problems of 50 within reasonable time. By analyzing the bottleneck, we found that 50% of the execution time are used to construct clauses for transitive closure, which can be easily parallelized. Also the synthesized result (i) does not starve any philosopher and (ii) ensures that each philosopher only needs to observe his left and right philosopher, making the resulting priority very desirable. Contrarily, it is possible to select a subset of components and ask to synthesize priorities for deadlock freedom using alphabet abstraction. The execution time using alphabet abstraction depends on the number of selected components; in our case we select 4 components thus is executed extremely fast. Of course, the synthesized result is not very satisfactory, as it starves certain philosopher. Nevertheless, this is unavoidable when overlooking interactions done by other philosophers. Except the traditional dining philosophers problem, we have also evaluated on (i) a BIP model (5 components) for data processing in digital communication (DPU; See Appendix A for description) (i) a simplified protocol of automatic traffic control (Traffic). Our preliminary evaluation on compositional priority synthesis is in Appendix B.

VII. Related Work

For deadlock detection, well-known model checking tools such as SPIN [18] and NuSMV [12] support deadlock detection by given certain formulas to specify the property. D-Finder [8] applies compositional and incremental methods to compute invariants for an over-approximation of reachable states to verify deadlock-freedom automatically. Nevertheless, all the above tools do not provide any deadlock avoidance strategies when real deadlocks are detected.

Synthesizing priorities is subsumed by the framework of controller synthesis proposed by Ramadge and Wohnham [22], where the authors proposed an automata-theoretical approach to restrict the behavior of the system (the modeling of environment is also possible). Essentially, when the environment is modeled, the framework computes the risk attractor and creates a centralized controller. Similar results using centralized control can be dated back from [5] to the recent work by Autili et al [4] (the SYNTHESIS tool). Nevertheless, the centralized coordinator forms a major bottleneck for system execution. Transforming a centralized controller to distributed controllers is difficult, as within a centralized controller, the execution of a local interaction of a component might need to consider the configuration of all other components.

[9]Available for download at http://www6.in.tum.de/~chengch/vissbip

[10]Evaluated under Intel 2.93GHz CPU with 2048Mb RAM for JVM.

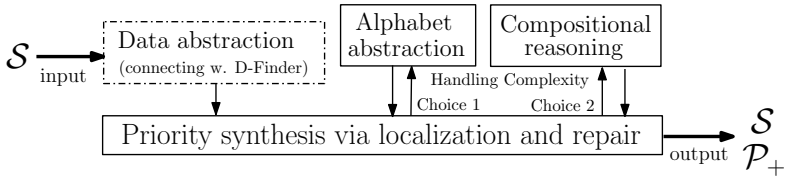

Figure 8. The framework of priority synthesis presented in this paper, where the connection with the D-Finder tool [8] is left for future work.

Priorities, as they are stateless, can be distributed much easier for performance and concurrency. E.g., the synthesized result of dining philosophers problem indicates that each philosopher only needs to watch his left and right philosophers without considering all others. We can continue with known results from the work of Graf et al. [17] to distribute priorities, or partition the set of priorities to multiple controllers under layered structure to increase concurrency (see work by Bonakdarpour et al. [9]). Our algorithm can be viewed as a step forward from centralized controllers to distributed controllers, as architectural constraints (i.e., visibility of other components) can be encoded during the creation of priority candidates. Therefore, we consider the work of Abujarad et al.[1] closest to ours, where they proceeds by performing distributed synthesis (known to be undecidable [21]) directly. In their model, they take into account the environment (which they refer it as faults), and consider handling deadlock states by either adding mechanisms to recover from them or preventing the system to reach it. It is difficult to compare two approaches directly, but we give hints concerning performance measure: (i) Our methodology and implementation works on game concept, so the complexity of introducing the environment does not change. (ii) In [1], for a problem of 10^{33} states, under 8-thread parallelization, the total execution time is 3837 seconds, while resolving the deadlock of the 50 dining philosophers problem (a problem of 10^{38} states) is solved within 31 seconds using our monolithic engine.

Lastly, the research of deadlock detection and mechanisms of deadlock avoidance is an important topic within the community of Petri nets (see survey paper [20] for details). Concerning synthesis, some theoretical results are available, e.g., [19], but efficient implementation efforts are, to our knowledge, lacking.

VIII. Conclusion

In this paper, we explain the underlying algorithm for priority synthesis and propose extensions to synthesize priorities for more complex systems. Figure 8 illustrates a potential flow of priority synthesis. A system can be first processed using data abstraction to create models suitable for our analysis framework. Besides the monolithic engine, two complementary techniques are available to further reduce the complexity of problem under analysis. Due to the stateless property and the fact that they preserve deadlock-freedom, priorities can be relatively easily implemented in a distributed setting.

Appendix

A. Data Processing Units in Digital Communication

In digital communication, to increase the reliability of data processing units (DPUs), one common technique is to use multiple data sampling. We have used VissBIP to model the components and synchronization for a simplified DPU. In the model, two interrupts (`SynchInt` and `SerialInt` respectively) are invoked sequentially by a `Master` to read the data from a `Sensor`. The `Master` may miss any of the two interrupts. Therefore, `SerialInt` records whether the interrupt from `SynchInt` is lost in the same cycle. If it is missed, `SerialInt` will assume that the two interrupts have read the same value in the two continuous cycles. According to the values read from the two continuous cycle, `Master` calculates the result. In case that the interrupt from `SerialInt` is missing in the second cycle or both interrupts are missing in the first cycle, `Master` will not calculate anything. Ideally, the calculation result from `Master` should be the same as what is computed in `SerialInt`. The mismatch will lead to global deadlocks.

The synthesis of VissBIP focuses on the deadlock-freedom property. First, we have selected the non-optimized engine. VissBIP reports that it fails to generate priority rules to avoid deadlock, in 4.174 seconds with 168 BDD variables. Then we have selected the optimized engine and obtained the same result in 0.537 seconds with 116 BDD variables. The reason of the failure is that two contradictory priority rules are collected in the synthesis. Finally, we have allowed the engine to randomly select a priority between the contradicts (priority-repushing). A successful priority is finally reported in 1.134 seconds to avoid global deadlocks in the DPU case study.

B. Compositional Priority Synthesis: A Preliminary Evaluation

Lastly, we conduct preliminary evaluations on compositional synthesis using dining philosophers problem. Due to our system encoding, when decomposing the philosophers problem to two subproblems of equal size, compare the subproblem to the original problem, the number of BDD variables used in the encoding is only 22.5% less. This is because the saving is only by replacing component construction with the assumption; for interactions, they are all kept in the encoding of the subsystem. Therefore, if the problem size is not big enough, the total execution time for compositional synthesis is not superior than than monolithic method, as the time spent on inappropriate assumptions

can be very costly. Still, we envision this methodology more applicable for larger examples, and it should be more applicable when the size of alphabet is small (but with lots of components).

REFERENCES

[1] F. Abujarad, B. Bonakdarpour, and S. Kulkarni. Parallelizing deadlock resolution in symbolic synthesis of distributed programs. In *Proceedings of the 8th International Workshop on Parallel and Distributed Methods in Verification (PDMC'09)*, volume 14 of *EPTCS*, pages 92–106, 2009.

[2] F. Aloul, I. Markov, and K. Sakallah. FORCE: a fast and easy-to-implement variable-ordering heuristic. In *Proceedings of the 13th ACM Great Lakes symposium on VLSI (GLSVLSI'03)*, pages 116–119. ACM, 2003.

[3] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987.

[4] M. Autili, P. Inverardi, A. Navarra, and M. Tivoli. SYNTHESIS: a tool for automatically assembling correct and distributed component-based systems. In *Proceedings of the 29th international conference on Software Engineering (ICSE'07)*, pages 784–787. IEEE Computer Society, 2007.

[5] S. Balemi, G. Hoffmann, P. Gyugyi, H. Wong-Toi, and G. Franklin. Supervisory control of a rapid thermal multiprocessor. *Automatic Control, IEEE Transactions on*, 38(7):1040–1059, 1993.

[6] A. Basu, M. Bozga, and J. Sifakis. Modeling heterogeneous real-time components in BIP. In *Proceedings of the 4th IEEE International Conference on Software Engineering and Formal Methods (SEFM'06)*, pages 3–12. IEEE, 2006.

[7] S. Bensalem, M. Bozga, J. Sifakis, and T. Nguyen. Compositional verification for component-based systems and application. In *Proceedings of the 6th International Symposium in Automated Technology for Verification and Analysis (ATVA'08)*, volume 5311 of *LNCS*, pages 64–79. Springer-Verlag, 2008.

[8] S. Bensalem, A. Griesmayer, A. Legay, T.-H. Nguyen, J. Sifakis, and R.-J. Yan. D-Finder 2: Towards Efficient Correctness of Incremental Design. In *Proceedings of the 3rd NASA Formal Methods Symposium (NFM'11)*, LNCS. Springer-Verlag, 2011.

[9] B. Bonakdarpour, M. Bozga, and J. Quilbeuf. Automated distributed implementation of component-based models with priorities. In *Proceedings of the 11th International conference on Embedded Software (EMSOFT'11)*, 2011. to appear.

[10] C.-H. Cheng, S. Bensalem, B. Jobstmann, R.-J. Yan, A. Knoll, and H. Ruess. Model construction and priority synthesis for simple interaction systems. In *Proceedings of the 3rd NASA Formal Methods Symposium (NFM'11)*, volume 6617 of *LNCS*, pages 466–471. Springer-Verlag, 2011.

[11] C.-H. Cheng, B. Jobstmann, C. Buckl, and A. Knoll. On the hardness of priority synthesis. In *Proceedings of the 16th International Conference on Implementation and Application of Automata (CIAA'11)*, volume 6807 of *LNCS*. Springer-Verlag, 2011.

[12] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new symbolic model verifier. In *Proceedings of the 11th Conference on Computer-Aided Verification (CAV'99)*, volume 1633 of *LNCS*, pages 495–499. Springer-Verlag, 1999.

[13] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT-Press, 1999.

[14] J. Cobleigh, D. Giannakopoulou, and C. Păsăreanu. Learning assumptions for compositional verification. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*, volume 2619 of *LNCS*, pages 331–346. Springer-Verlag, 2003.

[15] G. Gößler and J. Sifakis. Priority systems. In *Proceedings of the 2nd International Symposium on Formal Methods for Components and Objects (FMCO'03)*, volume 3188 of *LNCS*, pages 314–329. Springer-Verlag, 2003.

[16] E. Gradel, W. Thomas, and T. Wilke. *Automata, Logics, and Infinite Games*, volume 2500 of *LNCS*. Springer-Verlag, 2002.

[17] S. Graf, D. Peled, and S. Quinton. Achieving distributed control through model checking. In *Proceedings of the 22nd International Conference on Computer Aided Verification (CAV'10)*, volume 6174 of *LNCS*, pages 396–409. Springer-Verlag, 2010.

[18] G. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2004.

[19] M. Iordache, J. Moody, and P. Antsaklis. Synthesis of deadlock prevention supervisors using Petri nets. *Robotics and Automation, IEEE Transactions on*, 18(1):59–68, 2002.

[20] Z. Li, M. Zhou, and N. Wu. A survey and comparison of Petri net-based deadlock prevention policies for flexible manufacturing systems. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 38(2):173–188, 2008.

[21] A. Pneuli and R. Rosner. Distributed reactive systems are hard to synthesize. In *Proceedings of the 31st Annual Symposium on Foundations of Computer Science (FOCS'90)*, volume 0, pages 746–757 vol.2. IEEE Computer Society, 1990.

[22] P. Ramadge and W. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81–98, 1989.

[23] S. Tani, K. Hamaguchi, and S. Yajima. The complexity of the optimal variable ordering problems of shared binary decision diagrams. In *Algorithms and Computation*, volume 762 of *LNCS*, pages 389–398. Springer-Verlag, 1993.